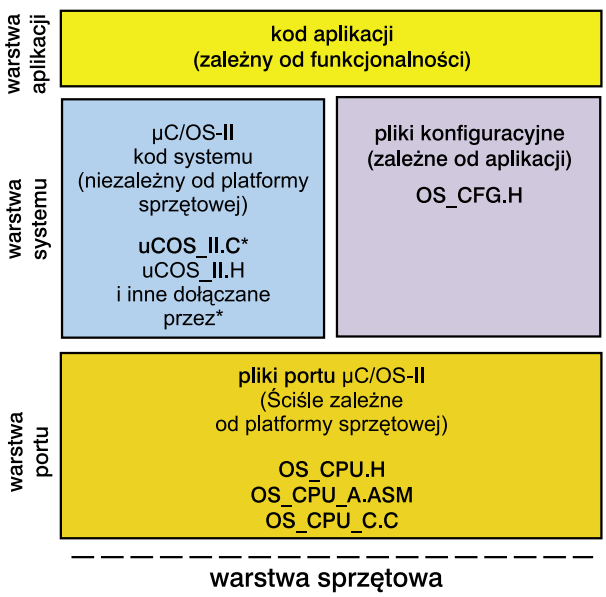
System operacyjny µC/OS–II na platformie AVR

µC/OS (Micro-Controller Operating System) to system operacyjny czasu rzeczywistego, napisany w języku ANSI C i przeznaczony głownie dla aplikacji wbudowanych (ang. embedded systems). Druga odsłona tego systemu, czyli µC/OS-II należy od lat do kanonu najlepiej udokumentowanych systemów tego typu, dlatego polecana jest szczególnie dla osób stawiających pierwsze kroki w programowaniu wielowątkowym. Nie oznacza to natomiast, że jest to system o małych możliwościach. Jego komercyjna wersja jest standardem w wielu profesjonalnych zastosowaniach, a liczba procesorów i mikrokontrolerów, dla których został on przystosowany jest imponująca i stale rośnie (obecnie ponad 40 rodzin).

Dzieje systemu µC/OS–II rozpoczęły się ponad 15 lat temu, kiedy to jego autor – Jean J. Labrosse, sfrustrowany działaniem dostępnych w owym czasie systemów operacyjnych czasu rzeczywistego (ang. *RTOS – Real Time Operating Systems*) postanowił napisać swój własny. I nie byłoby w tym może nic specjalnego, gdyby nie to, że zdecydował się opublikować swój kod razem z obszerną dokumentacją, w postaci książki [1], opisującej nie tylko jego system, ale także generalne mechanizmy oprogramowania tego typu. Szybko okazało się, że jest to jeden z najlepiej zaprojektowanych systemów na rynku. Jego ewolucja poszła w kierunku zwiększenia stabilności działania, bardziej niż dodawania nowych fajerwerków. To co wyróżnia go do dziś na tle konkurencji to naprawdę małe zużycie zasobów (moc obliczeniowa, pamięć), świetna dokumentacja oraz silnie deterministyczne działanie. Dzięki temu może być stosowany w aplikacjach wymagających bezkompromisowej pewności działania (ang. *Safety Critical Systems*), takich jak układy awioniki czy urządzenia medyczne. µC/OS–II został przystosowany do pracy z kilkudziesięcioma różnymi rodzinami procesorów i mikrokontrolerów, obejmującymi jednostki 8–, 16– i 32–bitowe. W artykule pokażemy sposób implementacji systemu na 8–bitowym mikrokontrolerze ATmega128 firmy Atmel, z wykorzystaniem narzędzi GNU.



Rys. 1. Warstwy oprogramowania w systemie µC/OS–II

Po co nam RTOS?

Zastosowanie systemu operacyjnego okupione jest zużyciem zasobów pamięci kodu i pamięci operacyjnej, a także pewnej mocy obliczeniowej (ok. 2 do 5% czasu CPU). Co otrzymujemy w zamian? Przede wszystkim łatwiejszą kontrolę nad tym co się dzieje w aplikacji (dotyczy to szczególnie zależności czasowych) oraz prostsze dodawanie nowej funkcjonalności. Dzięki zastosowaniu RTOS możemy podzielić aplikację na wątki (zadania), które będą wykonywane niezależnie, w kolejności zgodnej z przyznanym im priorytetem. System zadba o to, aby mikrokontroler w odpowiednich chwilach czasowych przełączał się z wykonywania jednego zadania na drugie. Oznacza to, że możemy pisać kod danego wątku tak, jakby tylko on miał się wykonywać. Więcej informacji na ten temat można znaleźć w literaturze [1–3].

Architektura oprogramowania wykorzystującego uC/OS–II

Portowanie oprogramowania (nie tylko systemu operacyjnego) na daną platformę sprzętową polega najczęściej na zaimplementowaniu pewnych specyficznych funkcji, działających w najniższej – najbliższej sprzętowi warstwie oprogramowania (**rys. 1**). µC/OS–II definiuje cały zestaw takich funkcji, przy czym nie wszystkie muszą być wykorzystywane.

Pliki portu implementują specyficzne funkcje, zdefiniowane przez system µC/OS–II, ale zależne od platformy sprzętowej. Jest to przede wszystkim obsługa stosu, zapis i odtwarzanie stanu rejestrów CPU, zarządzanie obsługą przerwań, w tym przerwania systemowego. Warstwa systemu zawiera implementację mechanizmów szeregowania zadań i różnych wariantów komunikacji międzywątkowej, alokacji pamięci, obsługi czasu systemowego itp. Najwyższą warstwą jest aplikacja użytkownika, obejmująca już konkretne zadania uruchamiane w systemie, które zapewniają wymaganą funkcjonalność.

List. 1.

```
#ifndef __ASSEMBLER__
typedef INT16U                OS_FLAGS;
#endif // __ASSEMBLER__
```

Co będzie nam potrzebne do rozpoczęcia pracy z µC/OS–II

Przede wszystkim musimy zaopatrzyć się w kod źródłowy µC/OS–II oraz ściągnąć pliki portu do ATmega128 ze strony producenta (*www.micrium.com*). Skorzystamy z portu dla mikrokontrolerów AVR i kompilatora GNU V3.xx. Będziemy więc potrzebować narzędzi do kompilacji i linkowania programów, a także samej edycji kodu – idealnie tę rolę spełnia dostępny darmowo pakiet WinAVR (*http://sourceforge.net/projects/winavr/*). Gotowy program trzeba będzie także skopiować do pamięci Flash mikrokontrolera wykorzystując jeden z dostępnych programatorów wraz z oprogramowaniem (np. darmowym AVR Studio).

Uruchomienie pierwszego programu wykorzystującego µC/OS–II może okazać się skomplikowane, dlatego warto wcześniej upewnić

List. 2.
```
void Task1(void *pdata)
{
    DDRB |= 0x01;          // ustaw pin 0 portu B jako wyjście
    while(1) {
        PORTB ^= 0x01;     // zmień stan portu
        OSTimeDly(120);    // wprowadź opóźnienie
    }
} /* Task1 */
```

List. 3.
```
void Task2(void *pdata)
{
    DDRB |= 0x02;    // ustaw pin 1 portu B jako wyjście
    while(1) {
        PORTB ^= 0x02;     // zmień stan portu
        OSTimeDly(60);     // wprowadź opóźnienie
    }
} /* Task2 */
```

się, że zarówno kompilator C, jak i programator działają prawidłowo. Kompilujemy i uruchamiamy w tym celu najprostszy program zmieniający stan na wyjściu portu, przez co zawężamy obszar poszukiwania ewentualnych błędów mogących powstać w późniejszym etapie.

Kody źródłowe wszystkich opisywanych w dalszej części programów oraz inne pożyteczne informacje można znaleźć na stronie *http://home.agh.edu.pl/~lkrzak* w dziale AVR.

Kompilujemy µC/OS–II

Zacznijmy od dodania niezbędnych plików do projektu. W tym celu w pliku *makefile* dodajemy jako kod źródłowy w C plik *UCOS_II.C*, który dołączy wszystkie wymagane przez warstwę systemu pliki. Warto wcześniej otworzyć ten plik i sprawdzić czy znajdujące się w nim ścieżki do pozostałych plików mają pokrycie w rzeczywistości. Dodatkowo, może się pojawić problem z dużymi i małymi literami w nazwach plików – narzędzia GNU są na to wrażliwe i konsekwentnie będą produkować błędy podczas kompilacji, jeżeli tego nie dopilnujemy. Do skryptu kompilacji musimy dołączyć także pliki warstwy portu. Są to *OS_CPU_C.C* oraz *OS_CPU_A.ASM*. Ten drugi musi być dołączony jako plik kodu w języku asembler. Niestety, w dostępnym na stronie producenta porcie uwidacznia się problem polegający na tym, że podczas przetwarzania pliku *OS_CPU_A.ASM* dołączany jest dyrektywą *#include* plik *OS_CFG.H*, w którym nieopatrznie umieszczono definicję typu *OS_FLAGS*. Asembler jej nie rozumie i zgłasza błąd. Można temu zapobiec dodając warunek dla preprocesora, który wyłączy ją podczas przetwarzania pliku przez asembler (**list. 1**).

Poprawnie napisany plik *makefile*, uwzględniający te poprawki powinien przeprowadzić „czystą” kompilację z minimalną liczbą ostrzeżeń, nawet przy ustawionym najbardziej rygorystycznym sprawdzaniu kodu (opcja kompilatora *–pedantic*). W testowanej wersji systemu, tj. 2.52 i kompilatorze w wersji 3.4.6 wykryto tylko jedno ostrzeżenie, wynikające z pozostawienia nieużywanej zmiennej w pliku *OS_TASK.C*.

Uruchamiamy system

Program działający pod kontrolą systemu operacyjnego jest dzielony na zadania (wątki), które są przełączane i działają pozornie niezależnie od siebie. W systemie zawsze działa przynajmniej jedno zadanie, tzw. wątek bezczynności (*idle task*) utworzony przez system. W systemach operacyjnych bez wywłaszczania zadań (*non–preemptive OS*) zadania same decydują o tym, kiedy przekazać kontrolę CPU innemu wątkowi. Komplikuje to tworzenie aplikacji, ponieważ programista sam musi zatroszczyć się o to, aby zadania w odpowiednim czasie zwolniły zasoby CPU. µC/OS–II jest natomiast systemem z wywłaszczaniem (*preemptive OS*), co oznacza, że sam decyduje o tym, które zadanie ma w danym czasie przejąć kontrolę. Konieczne jest więc okresowe „dopuszczenie do głosu” warstwy systemu, która jeżeli zajdzie taka konieczność, przełączy zadanie. Idealnym do tych celów jest mechanizm periodycznych przerwań, generowanych np. przez układ sprzętowego licznika (timera). Port dla ATmega128 wykorzystuje domyślnie do tego celu przerwanie od przepełnienia licznika TIMER0. Należy więc odpowiednio ustawić rejestry układu, aby przerwanie to było generowane. Poniższe dwie linie kodu w zasadzie załatwiają sprawę, przy założeniu, że przerwania są globalnie odblokowane.

```
TCCR0 = 0x06;
TIMSK = 0x01;
```

Dla częstotliwości taktowania mikrokontrolera wynoszącej 8 MHz otrzymujemy przerwanie co ok. 8 ms. Pamiętajmy, że im krótszy okres przerwania tym częściej przełączane mogą być zadania, a więc wątki o wyższym priorytecie szybciej są w stanie przejąć kontrolę. Niestety jednocześnie wzrasta tak-

List. 4.
```
void Task3(void *pdata)
{
    while ((PINB & 0x04)) {       // sprawdzaj pin w pętli
        OSTimeDly(2);
    };
    OSSemPost(event1);            // sygnalizuj zdarzenie event1
    OSTaskSuspend(OS_PRIO_SELF);  // wstrzymaj zadanie

} /* Task3 */
```

List. 5.
```
void Task2(void *pdata)
{
    INT8U error;
                                  // zaczekaj na zdarzenie
    OSSemPend(event1, 1000, &error);
    if (OS_NO_ERR == error) {
                                  // zdarzenie wystąpiło
        DDRB |= 0x02;             // ustawienie pinu jako wyjście
        while (1) {
            PORTB ^= 0x02;        // zmiana stanu portu
            OSTimeDly(60);
        }
    } else {
                                  // przekroczono czas oczekiwania na zdarzenie
        DDRB |= 0x02;             // ustawienie pinu jako wyjście
        PORTB &= 0x02;            // ustawienie niskiego stanu na porcie
        while (1) {
            ;
        }
    }
} /* Task2 */
```

że procentowe zużycie mocy obliczeniowej potrzebnej na częstsze działanie mechanizmu szeregowania zadań [2]. Precyzyjniejszą kontrolę nad czasem systemowym można uzyskać zmieniając typ przerwania z przepełnienia (*overflow*) na porównanie (*output compare*). Wymaga to niewielkiej ingerencji w pliki portu µC/OS–II.

Po skonfigurowaniu przerwania licznika pora zainicjalizować system, w tym celu wywołujemy funkcję *OSInit()*. Uruchomienie systemu następuje po wywołaniu funkcji *OSStart()*. Jeżeli wszystko poszło dobrze, program nie powinien już opuścić tej funkcji, jest ona bowiem główną pętlą systemową.

Migamy LED–ami – program 1

Spróbujmy napisać zadanie (**list. 2**), które będzie cyklicznie gasić i zapalać diodę LED podłączoną do pinu 0 portu B (zakładamy, że dioda świeci przy niskim stanie na pinie). Zadanie to będzie się składać z pętli, w której będziemy zmieniać stan na wyjściu portu, zaś operacja ta będzie przeplatana z funkcją opóźniającą zadanie o 120 taktów przerwania systemowego (ok. 1 s).

Drugie zadanie napiszemy tak, aby zmieniało stan pinu 1 portu B dwa razy szybciej (**list. 3**).

Mając dane funkcje, możemy je wprowadzić do systemu jako wątki. Wcześniej musimy jednak zapewnić im pewien obszar pamięci RAM na stos, potrzebny do zapisania kontekstu zadania (m.in. rejestry CPU i zmienne lokalne). W tym celu dla każdego zadania deklarujemy globalnie tablicę o domyślnym rozmiarze:

```
OS_STK Task1Stk[OS_TASK_DEF_STK_
SIZE];
OS_STK Task2Stk[OS_TASK_DEF_STK_
SIZE];
```

Teraz możemy już zainicjalizować zadania, przeznaczając je do uruchomienia poprzez umieszczenie przed *OSStart()* wywołań:

```
OSTaskCreate(Task1, NULL, (void
*)&Task1Stk[OS_TASK_DEF_STK_SIZE
- 1], 4);
OSTaskCreate(Task2, NULL, (void
*)&Task2Stk[OS_TASK_DEF_STK_SIZE
- 1], 5);
```

Zwróćmy uwagę na trzeci parametr tych funkcji, oznaczający wskaźnik do obszaru stosu zadania. W mikrokontrolerach AVR stos rośnie w kierunku młodszych adresów, dlatego jako początkową wartość wskaźnika stosu zadania podajemy adres ostatniego bajtu zadeklarowanej do tego celu wcześniej tablicy. W systemie µC/OS–II każde zadanie musi posiadać unikatowy priorytet. Nie dozwolone jest więc dzielenie czasu procesora na działanie dwóch zadań o tym samym priorytecie, jak ma to miejsce np. w systemie eCOS (tzw. *round–robin scheduling*). W naszym przypadku zadanie 2 będzie miało wyższy priorytet (=5) od zadania 1 (=4). Po skompilowaniu kodu i zaprogramowaniu mikrokontrolera obie diody powinny migać, przy czym jedna dwa razy szybciej od drugiej.

Dodajemy komunikację między zadaniami – program 2

Zmodyfikujmy nasz program tak, aby druga dioda zaczęła migać dopiero po pojawieniu się na pinie 2 portu B stanu niskiego. W przypadku, gdy sytuacja ta nie wystąpi, dioda po zadanym czasie oczekiwania zapali się na stałe. Stan pinu wejściowego będzie mo-

nitorowany przez trzecie zadanie, które po zakończeniu działania zostanie wstrzymane. Ponieważ zadania nie wiedzą nic o sobie, komunikację między nimi trzeba oprzeć o struktury globalne lub systemowe. µC/OS–II posiada mechanizmy dedykowane do komunikacji międzywątkowej. Są to generalnie semafory (*semaphores*) i skrzynki pocztowe (*mailboxes*). Dokładny opis tych mechanizmów wykracza poza ramy tego artykułu, spróbujmy więc posłużyć się przykładem wykorzystującym semafor.

Aby móc skorzystać z semafora trzeba go najpierw utworzyć wywołaniem:

```
OS_EVENT event1;  // zmienna glo-
balna
...
event1 = OSSemCreate(0);
```

nadając mu w ten sposób wartość początkową równą 0 i wiążąc go z globalną zmienną *event1*, która jest typu *OS_EVENT*. Utwórzmy wobec tego trzecie zadanie, podobnie jak dwa poprzednie (**list. 4**).

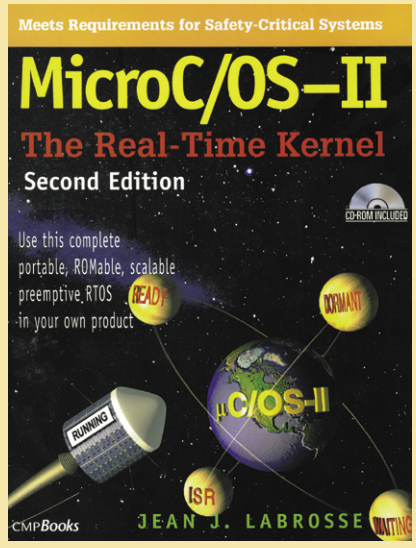
Zadanie to najpierw sprawdza w pętli czy nastąpiła zmiana stanu na wejściu. Jeśli tak, to sygnalizowane jest zdarzenie *event1*, co oznacza że wartość semafora zostanie zwiększona z zero na jeden. Następnie zadanie wstrzymuje swoje działanie.

Zmodyfikujmy zadanie 2 w ten sposób, aby czekało na zdarzenie *event1* (**list. 5**).

Funkcja *OSSemPend* czeka na wystąpienie zdarzenia *event1*. Jeżeli po czasie 1000 cykli systemowych zdarzenie nie wystąpi, to w zmiennej code pojawi się wartość *OS_TIMEOUT*, w przeciwnym razie będzie tam wartość *OS_NO_ERR* i dioda zacznie migać.

„Szyjemy" system na miarę

Działanie systemu można w pewnym stopniu dopasować do danej aplikacji, edytując plik *OS_CFG.H*. Znajduje się tam wiele definicji, wpływających na sposób działania systemu i zajmowane zasoby. W szczególności istnieje możliwość wyłączania nieużywanych funkcji lub całych modułów z procesu kompilacji, co zaoszczędzi nam cenną pamięć. Daną funkcję wykluczamy wpisując przy związanej z nią definicji zero. Przykładowo, jeżeli nie potrzebujemy funkcji kasującej utworzone zadania, wpisujemy:



Książka jest uznanym na świecie, kompletnym przewodnikiem po systemie operacyjnym µC/OS, napisanym przez jego autora. Doskonale spełnia rolę podręcznika dla programistów tworzących aplikacje na platformie µC/OS.

```
OS_TASK_DEL_EN     0   /* Include
code for OSTaskDel() */
```

W pierwszej aplikacji możemy skorzystać z domyślnych ustawień, warto jednak przyjrzeć się kilku definicjom:

`OS_CPU_HOOKS_EN` – jeśli równe 1, system będzie umożliwiał podłączenie specjalnych funkcji tzw. haków (ang. *hooks*), wywoływanych w specyficznych momentach, np. podczas przełączania zadania lub w trakcie pracy wątku bezczynności. Możemy je wykorzystać chociażby do monitorowania działania naszego programu.

`OS_TASK_STAT_EN` – jeśli równe 1, w tle działać będzie zadanie statystyczne, liczące procentowe wykorzystanie czasu CPU.

`OS_TASK_DEF_STK_SIZE` – domyślny rozmiar stosu dla zadań. W naszym programie korzystamy z tej definicji, aby przedzielić pamięć na stos obu zadań. Nie musi tak być. Lepiej z punktu widzenia zużycia pamięci byłoby dobrać rozmiar stosu do każdego zadania, biorąc pod uwagę liczbę zmiennych lokalnych, liczbę możliwych zagnieżdżeń itp. Analityczny dobór rozmiaru stosu jest zadaniem skomplikowanym, µC/OS–II posiada jednak mechanizmy pozwalające oszacować tę wielkość metodą testowania [1].

Podsumowanie

µC/OS–II zapewnia większość podstawowych mechanizmów potrzebnych w systemach czasu rzeczywistego, zużywając przy tym naprawdę niewiele zasobów. Idealnie nadaje się zarówno do systemów z małymi mikrokontrolerami, jak i większych systemów (nawet komputerów PC), w których muszą zostać spełnione ścisłe reżimy czasowe. Niezawodność systemu została potwierdzona wieloma certyfikatami, a także ogromną liczbą wdrożeń. Warto poznać µC/OS–II, zwłaszcza, jeśli ma to być pierwszy krok na drodze do programowania w aplikacjach wbudowanych z wykorzystaniem systemu operacyjnego czasu rzeczywistego.

µC/OS–II w wersji 2.52 jest rozpowszechniany w postaci kodu źródłowego dołączonego na płycie CD do książki pt. „MicroC/OS–II The Real–Time Kernel" napisanej przez autora systemu J. L. Labrosse'a. Książka ta stanowi wyjątkową pozycję, łączącą podręcznik użytkownika systemu z wnikliwym opisem najważniejszych mechanizmów występujących w systemach czasu rzeczywistego. Należy podkreślić, iż dystrybuowana wersja nie jest przeznaczona do celów komercyjnych – te wymagają osobnej (odpłatnej) licencji. Na koniec warto wspomnieć, iż producent systemu – firma Micrium ma w swojej ofercie biblioteki przeznaczone do zastosowania w systemach wbudowanych, obsługujące USB, stos TCP/IP, różne systemy plików, protokoły CAN i Modbus, itp.

Łukasz Krzak
Cezary Worek
Katedra Elektroniki AGH

Literatura

[1] J. L. Labrosse, „MicroC/OS–II The Real–Time Kernel", Second Edition, CMP Books 2002
[2] A. Lipowski, C. Worek, „Wprowadzenie do systemów operacyjnych dla systemów wbudowanych na przykładzie platformy ARM", Elektronika Praktyczna 11–12/2006
[3] A. Lipowski, C. Worek, „Systemy operacyjne w systemach mikroprocesorowych", Elektronika Praktyczna Plus 1/2006
[4] http://home.agh.edu.pl/~lkrzak